14｜常量：Go在“常量”设计上的创新有哪些？

2021-11-12 Tony Bai

讲述：Tony Bai
时长 17:32 大小 16.07M

你好，我是 Tony Bai。

在前面几节课中，我们学习了变量以及 Go 原生支持的基本数据类型，包括数值类型与字符串类型。这两类基本数据类型不仅仅可以被用来声明变量、明确变量绑定的内存块边界，还可以被用来定义另外一大类语法元素：**常量**。

领资料

你可能会问：常量有什么好讲的呢？常量不就是在程序生命周期内不会改变的值吗？如果是其他主流语言的常量，可讲的确实不多，但 Go 在常量的设计上是有一些“创新”的，都有哪些创新呢？我们不妨先来剧透一下。Go 语言在常量方面的创新包括下面这几点：

支持无类型常量；

支持隐式自动转型；

可用于实现枚举。

这些创新的具体内容是什么呢？怎么来理解 Go 常量的这些创新呢？你可以先思考一下，接下来我们再来详细分析。

不过在讲解这些“创新”之前，我们还是要从 Go 常量的一些基本概念说起，这会有助于我们对 Go 常量有一个更为深入的理解。

常量以及 Go 原生支持常量的好处

Go 语言的常量是一种在源码编译期间被创建的语法元素。这是在说这个元素的值可以像变量那样被初始化，但它的初始化表达式必须是在编译期间可以求出值来的。

而且，Go 常量一旦声明并被初始化后，它的值在整个程序的生命周期内便保持不变。这样，我们在并发设计时就不用考虑常量访问的同步，并且被创建并初始化后的常量还可以作为其他常量的初始表达式的一部分。

我们前面学过，Go 是使用 var 关键字声明变量的。在常量这里，Go 语言引入 const 关键字来声明常量。而且，和 var 支持单行声明多个变量，以及以代码块形式聚合变量声明一样，const 也支持单行声明多个常量，以及以代码块形式聚合常量声明的形式，具体你可以看下面这个示例代码：

复制代码

```
const Pi float64 = 3.14159265358979323846 // 单行常量声明

// 以const代码块形式声明常量
const (
    size int64 = 4096
    i, j, s = 13, 14, "bar" // 单行声明多个常量
)
```

不过，Go 语言规范规定，Go 常量的类型只局限于前面我们学过的 Go 基本数据类型，包括数值类型、字符串类型，以及只有两个取值（true 和 false）的布尔类型。

那常量的引入究竟给 Go 语言带来什么好处呢？没有对比便没有伤害。让我们先来回顾一下原生不支持常量的 C 语言的境况。

在 C 语言中，字面值担负着常量的角色，我们可以使用数值型、字符串型字面值来应对不同场合对常量的需求。

为了不让这些字面值以“魔数（Magic Number）”的形式分布于源码各处，早期 C 语言的常用实践是使用宏（macro）定义记号来指代这些字面值，这种定义方式被称为**宏定义常量**，比如下面这些宏：

复制代码

```
#define FILE_MAX_LEN 0x22334455
#define PI 3.1415926
#define GO_GREETING "Hello, Gopher"
#define A_CHAR 'a'
```

使用宏定义常量的习惯一直是 C 编码中的主流风格，即便后续的 C 标准中提供了 const 关键字后也是这样，但宏定义的常量会有很多问题。比如，它是一种仅在预编译阶段进行替换的字面值，继承了宏替换的复杂性和易错性，而且还有类型不安全、无法在调试时通过宏名字输出常量的值，等等问题。

即使我们改用后续 C 标准中提供的 const 关键字修饰的标识符，也依然不是一种圆满方案。因为 const 关键字修饰的标识符本质上依旧是变量，它甚至无法用作数组变量声明中的初始长度（除非用 GNU 扩展 C）。你可以看看下面这个代码，它就存在着这样的问题：

复制代码

```
const int size = 5;
int a[size] = {1,2,3,4,5}; // size本质不是常量，这将导致编译器错误
```

正是因为如此，作为站在 C 语言等编程语言的肩膀之上诞生的 Go 语言，它吸取了 C 语言的教训。Go 原生提供的用 const 关键字定义的常量，整合了 C 语言中宏定义常量、const 修饰的“只读变量”，以及枚举常量这三种形式，并消除了每种形式的不足，使得 Go 常量是类型安全的，而且对编译器优化友好。

Go 在消除了 C 语言无原生支持的常量的弊端的同时，还针对常量做了一些额外的创新。下面我们就来看第一个创新点：**无类型常量。**

无类型常量

通过前面的学习，我们知道 Go 语言对类型安全是有严格要求的：**即便两个类型拥有着相同的底层类型，但它们仍然是不同的数据类型，不可以被相互比较或混在一个表达式中进行运算。**这一要求不仅仅适用于变量，也同样适用于有类型常量（Typed Constant）中，你可以在下面代码中看出这一点：

复制代码

```
type myInt int
const n myInt = 13
const m int = n + 5 // 编译器报错：cannot use n + 5 (type myInt) as type int in

func main() {
    var a int = 5
    fmt.Println(a + n) // 编译器报错：invalid operation: a + n (mismatched types
}
```

而且，有类型常量与变量混合在一起进行运算求值的时候，也必须遵守类型相同这一要求，否则我们只能通过显式转型才能让上面代码正常工作，比如下面代码中，我们就必须通过将常量 n 显式转型为 int 后才能参与后续运算：

复制代码

```
type myInt int
const n myInt = 13
const m int = int(n) + 5  // OK

func main() {
    var a int = 5
    fmt.Println(a + int(n))  // 输出：18
}
```

那么在 Go 语言中，只有这一种方法能让上面代码编译通过、正常运行吗 ？当然不是，我们也可以使用 Go 中的无类型常量来实现，你可以看看这段代码：

复制代码

```
type myInt int
```

```
const n = 13

func main() {
    var a myInt = 5
    fmt.Println(a + n)  // 输出：18
}
```

你可以看到，在这个代码中，常量 n 在声明时并没有显式地被赋予类型，在 Go 中，这样的常量就被称为**无类型常量（Untyped Constant）**。

不过，无类型常量也不是说就真的没有类型，它也有自己的默认类型，不过它的默认类型是根据它的初值形式来决定的。像上面代码中的常量 n 的初值为整数形式，所以它的默认类型为 int。

不过，到这里，你可能已经发现问题了：常量 n 的默认类型 int 与 myInt 并不是同一个类型啊，为什么可以放在一个表达式中计算而没有报编译错误呢？

别急，我们继续用 Go 常量的第二个创新点，隐式转型来回答这个问题。

隐式转型

隐式转型说的就是，对于无类型常量参与的表达式求值，Go 编译器会根据上下文中的类型信息，把无类型常量自动转换为相应的类型后，再参与求值计算，这一转型动作是隐式进行的。但由于转型的对象是一个常量，所以这并不会引发类型安全问题，Go 编译器会保证这一转型的安全性。

我们继续以上面代码为例来分析一下，Go 编译器会自动将 a+n 这个表达式中的常量 n 转型为 myInt 类型，再与变量 a 相加。由于变量 a 的类型 myInt 的底层类型也是 int，所以这个隐式转型不会有任何问题。

不过，如果 Go 编译器在做隐式转型时，发现无法将常量转换为目标类型，Go 编译器也会报错，比如下面的代码就是这样：

复制代码

```
const m = 1333333333

var k int8 = 1
```

```
4  i := k + m // 编译器报错：constant 1333333333 overflows int8
```

这个代码中常量 m 的值 1333333333 已经超出了 int8 类型可以表示的范围，所以我们将它转换为 int8 类型时，就会导致编译器报溢出错误。

从前面这些分析中，我们可以看到，无类型常量与常量隐式转型的“珠联璧合”使得在 Go 这样的具有强类型系统的语言，在处理表达式混合数据类型运算的时候具有比较大的灵活性，代码编写也得到了一定程度的简化。也就是说，我们不需要在求值表达式中做任何显式转型了。所以说，在 Go 中，使用无类型常量是一种惯用法，你可以多多熟悉这种形式。

接下来，我们再来看看 Go 常量的最后一个重要创新，同样也是常量被应用较为广泛的一个领域：**实现枚举**。

实现枚举

不知道你有没有注意到，在前面讲解 Go 基本数据类型时，我们并没有提到过枚举类型，这是因为 **Go 语言其实并没有原生提供枚举类型**。

但是 Go 开发者对枚举这种类型的需求是现实存在的呀。那这要怎么办呢？其实，在 Go 语言中，我们可以使用 const 代码块定义的常量集合，来实现枚举。这是因为，枚举类型本质上就是一个由有限数量常量所构成的集合，所以这样做并没有什么问题。

不过，用 Go 常量实现枚举可不是我们的临时起意，而是 Go 设计者们的原创，他们在语言设计之初就希望将枚举类型与常量合二为一，这样就不需要再单独提供枚举类型了，于是他们将 Go 的前辈 C 语言中的枚举类型特性移植到常量的特性中并进行了“改良”。

那么接下来，我们就先来回顾一下 C 语言枚举类型，看看究竟它有哪些特性被移植到 Go 常量中了。在 C 语言中，枚举是一个命名的整型常数的集合，下面是我们使用枚举定义的 Weekday 类型：

复制代码

```
1 enum Weekday {
2     SUNDAY,
3     MONDAY,
4     TUESDAY,
```

```
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};

int main() {
    enum Weekday d = SATURDAY;
    printf("%d\n", d); // 6
}
```

你运行上面的 C 语言代码就会发现，其实 C 语言针对枚举类型提供了很多语法上的便利特性。比如说，如果你没有显式给枚举常量赋初始值，那么枚举类型的第一个常量的值就为 0，后续常量的值再依次加 1。

你看，上面这个代码中的 Weekday 枚举类型的所有枚举常量都没有显式赋值，那么第一个枚举常量 SUNDAY 的值就会被赋值为 0，它后面的枚举常量值依次加 1，这也是为什么输出的 SATURDAY 的值为 6 的原因。

但 Go 并没有直接继承这一特性，而是将 C 语言枚举类型的这种基于前一个枚举值加 1 的特性，分解成了 Go 中的两个特性：自动重复上一行，以及引入 const 块中的行偏移量指示器 iota，这样它们就可以分别独立使用了。

接下来我们逐一看看这两个特性。首先，**Go 的 const 语法提供了“隐式重复前一个非空表达式”的机制**，比如下面代码：

复制代码

```
const (
    Apple, Banana = 11, 22
    Strawberry, Grape
    Pear, Watermelon
)
```

这个代码里，常量定义的后两行并没有被显式地赋予初始值，所以 Go 编译器就为它们自动使用上一行的表达式，也就获得了下面这个等价的代码：

复制代码

```
const (
    Apple, Banana = 11, 22
```

```
    Strawberry, Grape  = 11, 22 // 使用上一行的初始化表达式
    Pear, Watermelon   = 11, 22 // 使用上一行的初始化表达式
)
```

不过，仅仅是重复上一行显然无法满足“枚举”的要求，因为枚举类型中的每个枚举常量的值都是唯一的。所以，**Go 在这个特性的基础上又提供了“神器”：iota**，有了 iota，我们就可以定义满足各种场景的枚举常量了。

iota 是 Go 语言的一个预定义标识符，它表示的是 const 声明块（包括单行声明）中，每个常量所处位置在块中的偏移值（从零开始）。同时，每一行中的 iota 自身也是一个无类型常量，可以像前面我们提到的无类型常量那样，自动参与到不同类型的求值过程中来，不需要我们再对它进行显式转型操作。

你可以看看下面这个 Go 标准库中 sync/mutex.go 中的一段基于 iota 的枚举常量的定义：

复制代码

```
// $GOROOT/src/sync/mutex.go
const (
    mutexLocked = 1 << iota
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota
    starvationThresholdNs = 1e6
)
```

这是一个很典型的诠释 iota 含义的例子，我们一行一行来看一下。

首先，这个 const 声明块的第一行是 mutexLocked = 1 << iota ，iota 的值是这行在 const 块中的偏移，因此 iota 的值为 0，我们得到 mutexLocked 这个常量的值为 1 << 0，也就是 1。

接着，第二行：mutexWorken 。因为这个 const 声明块中并没有显式的常量初始化表达式，所以我们根据 const 声明块里“隐式重复前一个非空表达式”的机制，这一行就等价于 mutexWorken = 1 << iota。而且，又因为这一行是 const 块中的第二行，所以它的偏移量 iota 的值为 1，我们得到 mutexWorken 这个常量的值为 1 << 1，也就是 2。

然后是 mutexStarving。这个常量同 mutexWorken 一样，这一行等价于 mutexStarving = 1 << iota。而且，也因为这行的 iota 的值为 2，我们可以得到 mutexStarving 这个常量的值为 1 << 2，也就是 4;

再然后我们看 mutexWaiterShift = iota 这一行，这一行为常量 mutexWaiterShift 做了显式初始化，这样就不用再重复前一行了。由于这一行是第四行，而且作为行偏移值的 iota 的值为 3，因此 mutexWaiterShift 的值就为 3。

而最后一行，代码中直接用了一个具体值 1e6 给常量 starvationThresholdNs 进行了赋值，那么这个常量值就是 1e6 本身了。

看完这个例子的分析，我相信你对于 iota 就会有更深的理解了。不过我还要提醒你的是，位于同一行的 iota 即便出现多次，多个 iota 的值也是一样的，比如下面代码：

复制代码

```
const (
    Apple, Banana = iota, iota + 10 // 0, 10 (iota = 0)
    Strawberry, Grape // 1, 11 (iota = 1)
    Pear, Watermelon  // 2, 12 (iota = 2)
)
```

我们以第一组常量 Apple 与 Banana 为例分析一下，它们分为被赋值为 iota 与 iota+10，而且由于这是 const 常量声明块的第一行，因此两个 iota 的值都为 0，于是就有了“Apple=0, Banana=10”的结果。下面两组变量分析过程也是类似的，你可以自己试一下。

如果我们要略过 iota = 0，从 iota = 1 开始正式定义枚举常量，我们可以效仿下面标准库中的代码：

复制代码

```
// $GOROOT/src/syscall/net_js.go
const (
    _ = iota
    IPV6_V6ONLY  // 1
    SOMAXCONN    // 2
    SO_ERROR     // 3
)
```

在这个代码里，我们使用了空白标识符作为第一个枚举常量，它的值就是 iota。虽然它本身没有实际意义，但后面的常量值都会重复它的初值表达式（这里是 iota），于是我们真正的枚举常量值就从 1 开始了。

那如果我们的枚举常量值并不连续，而是要略过某一个或几个值，又要怎么办呢？我们也可以借助空白标识符来实现，如下面这个代码：

复制代码

```
const (
    _ = iota // 0
    Pin1
    Pin2
    Pin3
    _
    Pin5    // 5
)
```

你可以看到，在上面这个枚举定义中，枚举常量集合中没有 Pin4。为了略过 Pin4，我们在它的位置上使用了空白标识符。

这样，Pin5 就会重复 Pin3，也就是向上数首个不为空的常量标识符的值，这里就是 iota，而且由于它所在行的偏移值为 5，因此 Pin5 的值也为 5，这样我们成功略过了 Pin4 这个枚举常量以及 4 这个枚举值。

而且，iota 特性让我们维护枚举常量列表变得更加容易。比如我们使用传统的枚举常量声明方式，来声明一组按首字母排序的“颜色”常量，也就是这样：

复制代码

```
const (
    Black  = 1
    Red    = 2
    Yellow = 3
)
```

假如这个时候，我们要增加一个新颜色 Blue。那根据字母序，这个新常量应该放在 Red 的前面呀。但这样一来，我们就需要像下面代码这样将 Red 到 Yellow 的常量值都手动加 1，十分费力。

复制代码

```
const (
    Blue   = 1
    Black  = 2
    Red    = 3
    Yellow = 4
)
```

那如果我们使用 iota 重新定义这组“颜色”枚举常量是不是可以更方便呢？我们可以像下面代码这样试试看：

复制代码

```
const (
    _ = iota
    Blue
    Red
    Yellow
)
```

这样，无论后期我们需要增加多少种颜色，我们只需将常量名插入到对应位置就可以，其他就不需要再做任何手工调整了。

而且，如果一个 Go 源文件中有多个 const 代码块定义的不同枚举，每个 const 代码块中的 iota 也是独立变化的，也就是说，每个 const 代码块都拥有属于自己的 iota，如下面代码所示：

复制代码

```
const (
    a = iota + 1 // 1, iota = 0
    b            // 2, iota = 1
    c            // 3, iota = 2
)

const (
    i = iota << 1 // 0, iota = 0
    j             // 2, iota = 1
    k             // 4, iota = 2
)
```

你可以看到，每个 iota 的生命周期都始于一个 const 代码块的开始，在该 const 代码块结束时结束。

小结

好了，今天的课讲到这里就结束了。今天我们学习了 Go 中最常用的一类语法元素：**常量**。

常量是一种在源码编译期间被创建的语法元素，它的值在程序的生命周期内保持不变。所有常量的求值计算都是在编译期完成的，而不是在运行期，这样可以减少运行时的工作，也方便编译器进行编译优化。另外，当操作数是常量表达式时，一些运行时的错误也可以在编译时被发现，例如整数除零、字符串索引越界等。

Go 语言原生提供了对常量的支持，所以我们可以避免像 C 语言那样，使用宏定义常量，这比较复杂，也容易发生错误。而且，Go 编译器还为我们提供的类型安全的保证。

接着，我们也学习了无类型常量，这是 Go 在常量方面的创新。无类型常量拥有和字面值一样的灵活性，它可以直接参与到表达式求值中，而不需要使用显式地类型转换。这得益于 Go 对常量的另一个创新：**隐式转型**，也就是将无类型常量的默认类型自动隐式转换为求值上下文中所需要的类型，并且这一过程由 Go 编译器保证安全性，这大大简化了代码编写。

此外，Go 常量还“移植”并改良了前辈 C 语言的枚举类型的特性，在 const 代码块中支持自动重复上一行和 iota 行偏移量指示器。这样我们就可以使用 Go 常量语法来实现枚举常量的定义。并且，基于 Go 常量特性的枚举定义十分灵活，维护起来也更为简便。比如，我们可以选择以任意数值作为枚举值列表的起始值，也可以定义不连续枚举常量，添加和删除有序枚举常量时也不需要手工调整枚举的值。

思考题

今天我也给你留了思考题：虽然 iota 带来了灵活性与便利，但是否存在一些场合，在定义枚举常量时使用显式字面值更为适合呢？你可以思考一下，欢迎在留言区留下你的答案。

感谢你和我一起学习，也欢迎你把这门课分享给更多对 Go 语言学习感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 基本数据类型：为什么Go要原生支持字符串类型？

下一篇 15 | 同构复合类型：从定长数组到变长切片

更多学习推荐

精选留言 (15)

写留言

qinsi
2021-11-12

文中颜色的例子不会有问题吗？因为颜色名称都是大写字母开头，所以有可能是公开API的一部分。像例子中那样在已有颜色前插入新的颜色的话会改变已有颜色的值，如果API的使用者没有跟着更新的话……

就好比定义protobuf消息时，发布出去的字段编号就不应该去修改了。

展开

作者回复: “像例子中那样在已有颜色前插入新的颜色的话会改变已有颜色的值” -- 这样的环节可能存在于api发布前的active development环节。

如果是api发布后，要插入颜色值，那么就要在api版本上做文章了。

另外这里仅是一个demo，为的是配合课程讲解。

1

qinsi
2021-11-12

所谓的“无类型”常量其实是有默认类型的，正因为有类型才会需要隐式的“类型转换”

1

liaomars
2021-11-12

在那种定义 比较大的数值的时候，比如：8906，"ADMIN" 这种常量值用字面量比较符合

共 1 条评论 >

1

高雪斌
2021-11-12

猛一看iota还以为是类似C++填充序列用的，原来只是个常量计数器，配合简单的数学表达式生成所期望的枚举或常量/枚举值

展开

1

Geek_b6da5d
2021-11-12

颜色rgba，各种size等，都需要显式字面值

展开

作者回复: 可以的。

1

用0和1改变自己
2021-11-12

当枚举常量是需要固定值，且业务上不可更改时，用显式字面值，这样可避免后续更改(插入或删除）带来的麻烦，这样的想法对吗？

展开

作者回复: 思路基本正确。

1

罗杰

2021-11-12

没有明显规则的枚举常量 iota 可能用起来就非常不舒服。看完这节我好像用了假的 Go 语言，我的笔记都是这可以吗？这编译不过吧，特别是隐式转换。老师真是 Go 语言百科全书！

1

将军

2021-11-12

个人觉得iota在一些场景下是方便一些，但更关键的在于代码可读性要好。像标准库syscall下有大量定义的常量，如果使用iota会看得头晕，心智负担太大。

展开

作者回复: 没错，也有一些场合不适于使用iota。你提到的syscall场合算是一个。因为系统调用的编号以及错误码值几乎形成了标准，直接用原值更好，也方便查找。

1

DullBird

2021-11-14

比如错误码，经常可能根据业务的情况删除不用的错误码。增加新的错误码？有动态增删情况的场景这样定义常量不好。适合比较固定。或者只增加的情况

展开

作者回复: 提醒一下，如果错误码随着api已经发布，那么还是尽量不要删除，可以标识作废。否则api就是不兼容的，需要升级api大版本。

aoe

2021-11-13

iota实现开闭原则很轻松啊

展开

2021-11-13

没想到小小的常量，在go里面居然那么多特点。
iota看起来挺神奇的，但其实习惯都显式写出来的我们似乎没那么吸引。而且，想一些非数值型的，iota显然也不那么胜任，比如颜色值、比例等~

Sam Fu

2021-11-13

老师还是多讲一下底层原理和最佳实践吧。itoa那种从没见人使用过。

> 作者回复: 无论是go标准库还是一些知名开源go项目，iota都有广泛使用的。这是go惯用法，建议多多关注。

lesserror

2021-11-13

Tony Bai 老师的每篇文章都很精彩。以下困惑，麻烦有空解答一下：

1. 文中说：“但由于转型的对象是一个常量，所以这并不会引发类型安全问题，Go 编译器会保证这一转型的安全性。”我理解的是Go编译器在编译阶段就能发现类型错误的问题，也就从而保证了类型转换的安全性吧。因为常量的类型的确定和运算都是在编译期间完...

展开

> 作者回复: 1. 差不多
> 2. iota的确是一个单词，表示“极微小”的意思。标准读音是 [aɪ'oʊtə] . 不过你也知道，很多程序员界的术语都读乱了，就像nginx这个词读啥的都有。
> 3. 篇幅有限，位运算没列入专题讲解。

1

功夫熊猫

2021-11-13

当你定义的值是一些固定值，比如端口或者进城的时候。或者就是那个数特别大

作者回复: good。

酥宝话不多

2021-11-12

状态码吗？

展开